Marzo 1999

F. Spagna

DATABASE IN RETE E SERVLET

F. Spagna

DATABASE IN RETE E SERVLET

1.1 Architettura client/server

Negli anni 60 il processamento dell'informazione era fatto principalmente sui calcolatori detti mainframe, che erano macchine grandi e costose. Poi negli anni 70 vennero i minicomputer e il timesharing, ma il modello era sempre quello centralizzato dei mainframe (è quello che possiamo chiamare *single tier*). Poi negli anni 80 arrivarono i personal computer e l'informazione fu frammentata su molteplici piccole macchine incostistenti tra di loro e poco sicure.

C'è stata quindi l'evoluzione verso l'architettura client/server che ha permesso di avere allo stesso tempo un controllo sui dati centralizzati (su uno o più server), che permetteva di mantenere la consistenza dei dati e un'accessibilità ad essi distribuita agli utenti (client) secondo i loro bisogni. Si è arrivati così alla soluzione *two-tier* nella quale i dati sono immagazzinati in un unico posto (server) e quindi comodi da maneggiare e consistenti, ma da parte dei client c'è una grande flessibilità in quanto il software di accesso può essere modificato a seconda delle esigenze. Dal lato server ci sono le due soluzioni di *database server* e *application server*: nel primo caso i metodi vengono eseguiti sul client ed il server è usato principalmente per immagazzinare o leggere i dati (database), mentre nel secondo i metodi sono eseguiti sul server in seguito alla richiesta fatta dal client tramite un protocollo di conversazione. Sia l'una sia l'altra soluzione possono essere migliorate con l'adozione di altri strati intermedi di software (*tier*): ad esempio in una soluzione *three-tier* esiste un componente intermedio tra il client ed il server, ma i sistemi client/server più in generale possono consistere in un numero qualsiasi di *tier* (*n-tier*), ciascuno strato fornendo un diverso livello di servizio tra lo stato superiore e quello inferiore con un miglioramento della flessibilità e dell'interoperabilità.

Con la progressione dei linguaggi orientati agli oggetti anche il client/server si è evoluto in direzione degli oggetti (componenti): in un'architettura a componenti possono interagire oggetti software in applicazioni diverse, che possono anche essere scritti in linguaggi diversi e girare su macchine diverse.

([11] JavaWorld october 1998)

1.2 Connessione ai database in Java

1.2.1 Il linguaggio SQL di interrogazione dei database

Il linguaggio SQL (*Structured Query Language*) è un linguaggio utilizzato per fare operazioni di lettura o scrittura sui database. Ciascun database ha la sua versione di SQL, anche se i diversi SQL sono simili tra di loro. Infatti, nonostante già dal 1992 esiste lo standard SQL-

92, i diversi database non sono ancora totalmente compatibili con quello standard, più che per l'accesso diretto, che è generalmente piuttosto stndardizzato, per le query che incrociano i dati di più tabelle.

[11.x] Marco Russo, Quanto sono compatibili con SQL 92 i DBMS, Computer Programming N.83, Settembre 1999, pag.32.

1.2.2 Java DataBase Connectivity (JDBC)

Il linguaggio Java dispone della libreria standard di sistema **JDBC** *(Java DataBase Connectivity)*, introdotta nella versione 1.1 del JDK, che permette l'accesso dalle applicazioni ai database relazionali[1] Sql. La connessione e l'interazione delle classi di questa libreria con i database, siano essi locali o remoti, avviene tramite dei **driver** specifici per ciascun particolare sistema di database, che sono disponibili per quasi tutti i database, e con l'uso del linguaggio SQL (nella versione ANSI-SQL 2 Entry Level) per l'interrogazione mediante *query*. JDBC permette di scrivere applicazioni indipendenti non solo dalla piattaforma come sono quelle Java, ma anche dal database, in quanto le classi sono svincolate dalle particolarità del database, di cui si prende carico il driver.

Il JDBC è conforme allo standard X/Open SQL Call Level Interface (CLI), come l'ODBC (*Open DataBase Coinnectivity*)[2] di Microsoft, già molto diffuso (quasi tutti i database ne hanno un'interfaccia), il quale però è stato sviluppato per il linguaggio C (con uso di puntatori e non ancora ad oggetti) e perciò non adatto al Java.

Il JDBC può utilizzare diversi tipi di driver:

- **driver nativi**, direttamente sviluppati per Java (e scritti essi stessi in Java), che trasformano le chiamate JDBC in chiamate native dirette del database;

- possono essere usati anche i **driver ODBC**, già largamente disponibili per tutti i database, tramite un ponte *(bridge)*, sviluppato da Intersolv in collaborazione con JavaSoft, che, utilizzando una DLL, la `jdbcodbc.dll`, trasforma le chiamate JDBC in chiamate per il driver ODBC, e rende quindi subito disponibili per Java tutti i database che possiedono già un'interfaccia ODBC, ma le prestazioni di questi driver sono peggiori di quelle dei driver diretti;

- un terzo tipo di driver, più flessibile, è costituito da un *gateway* che traduce le chiamate JDBC in un protocollo di rete indipendente dal database e lo comunica ad un server intermedio: è il caso del "DataGateway for Java" di Borland;

- il *"three tiered JDBC Remote Invocation Access"* è un sistema che richiede un terzo strato di software costituito da un'applicazione che gira sul server e comunica da una parte con l'applet del client e dall'altra con il database server.

Le classi e le interfacce del JDBC, facenti parte del package **`java.sql`**, supportano funzionalità di socket e permettono di fare operazioni come quella di caricare il driver specifico del database, aprire una connessione con il database, creare dei comandi SQL di interrogazione *(query)*, inviarli al database, riceverne i risultati, ottenere informazioni sulla struttura dei database, effettuare transazioni e chiudere la connessione. Le principali classi e interfacce che

[1] I database relazionali sono costituiti da tabelle in relazione tra loro.

[2] ODBC è un'interfaccia di programmazione costituita da uno strato di codice che consente alle applicazioni di accedere in modo uniforme ai dati in qualunque sistema di gestione di database che utilizza il linguaggio SQL come standard di accesso ai dati. Un driver ODBC per un certo database mappa ogni comando SQL sulle operazioni dello specifico database.

sovraintendono a queste operazioni sono: la classe **DriverManager** per il caricamento del driver e la connessione al database, la classe **Connection** che rappresenta la sessione di collegamento al database, l'interfaccia **Statement** per la creazione e l'esecuzione delle *query* e l'interfaccia **ResultSet** che contiene i risultati di un'interrogazione.

L'apertura della connessione viene fatta dalla classe `DriverManager` attraverso il suo metodo `getConnection()` che riceve come parametro una particolare forma di tipo JDBC dell'URL del database che contiene anche il nome del driver ed è così costituita:

```
jdbc:<driver>:<locazione>
```

dove la forma con cui è espressa la locazione dipende dal driver usato in quanto ciascun driver ha il suo proprio modo di esprimere la locazione.

Per esempio per un database di tipo miniSQL (o mSQL), il cui driver si chiama `msql`, la locazione ha la forma:

```
//<hostname>:<port>/<nome database>
```

e quindi si scriverebbe ad esempio:

```
jdbc:msql://bologna.enea.it:80/elenco
```

mentre per un database cui si accede tramite ODBC si scriverebbe ad esempio:

```
jdbc:odbc:<dsn>
```

ove il `dsn` è il *Data Source Name* che individua il driver e localizza nel sistema il database, che può essere costituito da un file locale o da un database server locale o remoto, essendo il driver installato localmente (sul sistema client).

Un quadro più completo delle interfacce e delle classi per la connessione ai database e l'esecuzione di statement SQL che Java fornisce è presentato qui di seguito:

Interfacce:

```
CollableStatement
Connection
DatabaseMetaData
Driver
PreparedStatement
ResultSet
ResultSetMetaData
Statement
```

Classi:

```
Data
DriverManager
DriverPropertyInfo
```

```
Time
Timestamp
Types
```

Facciamo un esempio completo di codice per l'accesso e l'interrogazione di un database di tipo ODBC tramite il ponte jdbc.odbc, ammettendo di avere sul sistema un database definito come `dati`:

```
// K01database.java (F.Spagna) Accesso a database con JDBC:ODBC

import java.awt.*;                                    // per l'interfaccia grafica
import java.sql.*;        // per DriverManager, Connection, Statement, ResultSet

public class K01database extends java.applet.Applet {

   String s[] = new String[1000];          // stringhe dei record del risultato
   int nRecord = 0;                          // numero di record del risultato
                                          // costruttore che apre la connessione,
   public K01database() {                 // fa la query SQL e riceve i risultati
                    try {
      // new sun.jdbc.odbc.JdbcOdbcDriver();                   // carica il driver
      DriverManager.setLogStream(null);
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection("jdbc:odbc:dati");
      Statement st = con.createStatement();
      String quer = "SELECT * FROM rubrica WHERE cognome LIKE 'P%'";  // query
      ResultSet ris = st.executeQuery(quer); // risultato della query eseguita
      while (ris.next())    // scorre tutti i risultati (finche' ci sono dati)
         System.out.println(s[nRecord++] = " " + nRecord + " " +
                            ris.getString("cognome")      + " " + // vari campi
                            ris.getString("nome")         + " " +
                            ris.getString("citta1")       + " " +
                            ris.getString("telefono1"));
      con.close();                                     // chiude la connessione
                    } catch(ClassNotFoundException e) {
                         System.out.println("Errore: " + e.getMessage()); }
                      catch(SQLException e) {
                         System.out.println("Errore: " + e.getMessage()); }
   }
   public void paint(Graphics g) {              // scrive i risultati se applet
      for (int n = 0; n < nRecord; n++)
         g.drawString(s[n], 10, 14*(n+1));
   }
   public static void main(String[] str) {     // metodo per applicazione Java
      new K01database();
   }
}
```


[11.01] M.Sciabarrà. Java: il JDBC, Computer Programming, settembre 1997, pag. 81.

[11.02] www.borland.com/datagateway/papers/datagateway/

1.2.3 Object-relational mapping

Sun Microsystems ha annunciato il **Java Blend**, prodotto basato su JDBC, costituito da un ambiente di sviluppo e da librerie, basato sulla cosiddetta *object-relational mapping* (traduzione e mappatura automatica dei dati del database in oggetti Java), che permette lo sviluppo di applicazioni Java che accedono ai database senza l'uso dell'SQL.

[11.03] da: http://www.javaworld.com/javaworld/jw-10-1997/jw-10-newsbriefs.html#12

[11.04] da: http://java.sun.com/products/java-blend/

1.2.4 DataGateway for Java

DataGateway for Java è una soluzione *multi-tier* per la connessione nativa ai database Oracle, DB2, Sybase, MS SQL Server, Informix, InterBase, MS Access, FoxPro, Paradox e dBase tramite un gateway su server NT.referenze#

1.3 Remote Method Invocation (RMI)

1.3.1 Applicazioni distribuite e RMI

Il **Remote Method Invocation** (conosciuto anche col suo acronimo **RMI**) permette ad un oggetto di un programma Java che gira su una JVM di comunicare con altri oggetti che risiedono su un'altra macchina virtuale (sullo stesso sistema locale o su un sistema remoto) e invocare i metodi.

Le applicazioni RMI sono applicazioni distribuite costituite da due programmi separati: uno server ed uno client. L'applicazione server crea degli oggetti (remoti per il client) di cui rende le referenze accessibili e resta in attesa dell'invocazione dei suoi metodi da parte di clienti, mentre l'applicazione client ottiene la referenza di oggetti remoti server ed invoca i loro metodi. L'RMI fornisce il meccanismo di comunicazione server/client in un'architettura del genere (ad "oggetti distribuiti"). Le operazioni che sono effettuate per mezzo di questa comunicazione sono:

- la localizzazione degli oggetti remoti e l'ottenimento delle loro referenze,

- la comunicazione tra gli oggetti remoti che appare come un'invocazione standard di metodi,

- il caricamento del bytecode (chiamato anche semplicemente *code*) di classi con il quale gli oggetti possono essere trasmessi, così che diventano dinamicamente presenti sulla JVM di arrivo in cui non fosse definita la loro classe (l'applicazione remota viene ad essere estesa dinamicamente).

Gli ***oggetti remoti*** sono oggetti che hanno metodi che possono essere invocati da una VM all'altra. Perché un oggetto sia remoto deve implementare un'interfaccia remote, che è un interfaccia derivata per estensione dall'interfaccia `java.rmi.Remote` e solo i metodi di quest'interfaccia sono disponibili per un'invocazione remota. Un oggetto remoto è trattato diversamente da un oggetto non remoto da RMI: infatti gli oggetti remoti quando sono passati da una VM all'altra non sono passati come copie, ma come stub, che funziona come un rappresentante locale dell'oggetto remoto ed è un'effettiva referenza all'oggetto remoto: quando viene invocato un metodo dello stub locale questo si occupa di trasferire questa chiamata sull'oggetto remoto. Uno stub implementa le stesse interfacce dell'oggetto remoto che rappresenta. Gli stub degli oggetti remoti di un'applicazione distribuita sono creati con il compilatore **`rmic`**.

Le fasi della creazione di un'applicazione RMI possono essere:

- definire gli oggetti che devono essere accessibili in modo remoto e non solo locali,

- definire le interfacce remote che contengono i metodi che possono essere invocati remotamente da un applicazione client,

- implementare le classi degli oggetti remoti con le relative interfacce remote (anche quelle relative a oggetti da usare come parametri o valori di ritorno di metodi),

- compilare con `javac` le interfacce e le applicazioni server e client con le loro classi,

- usare il compilatore `rmic` per creare gli stub degli oggetti remoti,

- rendere accessibile tutto quello che serve,

- far partire l'*RMI remote object registry* e le applicazioni server e client.

1.3.2 Esempio di applicazione RMI

Facciamo adesso un esempio di creazione di un'applicazione distribuita che utilizza l'RMI.

[] Ann Wollrath, Jim Waldo, An Overview of RMI Applications, http://java.sun.com/docs/books/tutorial/rmi/overview.html

1.4 Il linguaggio Java nei server Web

1.4.1 Le tecniche server preesistenti: i programmi CGI

Con la tecnica CGI (*Common Gateway Interface*) un server Web esegue un'applicazione (programma detto appunto CGI) in corrispondenza di un richiamo specifico ad esso fatto sull'HTML tramite un form che raccoglie i dati di input da un cliente e li trasmette al server chiedendogli di lanciare un'applicazione (*action*). L'applicazione genera un output in formato HTML che viene inviato al cliente come risposta. I programmi CGI (detti generalmente scripts) possono essere degli eseguibili compilati (per la maggior parte scritti in linguaggio C/C++) o anche script interpretati, per esempio in linguaggio Perl.

Dal punto di vista delle prestazioni le applicazioni CGI sono poco efficienti: infatti per ogni richiesta esse richiedono al sistema operativo di aprire, ed alla fine chiudere, un nuovo processo e procedono ogni volta all'apertura delle connessioni richieste, ad esempio con database.

Nei server Web più evoluti si adottano soluzioni specifiche per superare questo inconveniente con applicazioni server compilate sotto forma di DLL (*Dynamic Link Library*) e caricate sullo stesso spazio di memoria del server. In questo modo l'esecuzione dell'applicazione non va ad aprire ogni volta un nuovo processo, ma si riduce ad una semplice chiamata di funzione, e così le prestazioni migliorano molto, ma in questo modo le applicazioni perdono ogni portabilità in quanto sono legate allo specifico server Web, dal momento che i vari server adottano soluzioni particolari con standard diversi (per l'Internet Information Server o IIS di Microsoft lo standard è l'ISAPI e per i server di Netscape lo NSAPI).

[11.] Greg Alwang, Internet Web Servers, PC Magazine May 5, 1998, page 184.

[11.] Jeff Downey, Rise above the Static, PC Magazine May 5, 1998, page 271.

1.4.2 Java sui server Web

L'utilizzazione del linguaggio Java per applicazioni Web non è limitato alla creazione di applet, che, ricordiamo, sono applicazioni (generalmente di dimensioni non eccessive per ragioni di tempi di *download*) provenienti da un server ed eseguite e visualizzate sulla macchina del cliente, ma Java può anche essere adoperato per programmare applicazioni, che, come i CGI, vanno in esecuzione sul server in seguito alla richiesta di un client, costituendo la componente server di applicazioni client/server Internet o intranet.

Le ragioni per cui il Java si presta molto bene alla programmazione della parte server di applicazioni Web sono in gran parte le stesse di quelle che fanno di Java un linguaggio di elezione per le applicazioni Web lato cliente (applet), e tra esse ricordiamo principalmente l' *object-orientation*, l'indipendenza dalla piattaforma e la portabilità del codice (che consentono l'adozione di un unico linguaggio di sviluppo, con gran vantaggio per chi lavora su server di tipi diversi), la sicurezza insita nel linguaggio stesso (divieto di accesso diretto alla memoria, eliminazione dei puntatori, *type-checking* stretto con l'obbligatorietà del casting di tipo) e il suo controllo tramite un *Security Manager*, che permette una gestione dei permessi particolareggiata non possibile con altri sistemi, il supporto del multithreading nativo, la gestione della memoria insita nel linguaggio e l'eliminazione, con la *garbage collection*, dei rischi di *memory likage* (cioè di memoria non disallocata quando non è più usata) e ancora l'*exception handling*. Per non parlare del deciso miglioramento delle prestazioni rispetto alle applicazioni CGI.

Le **servlet** sono oggetti Java (e in quanto tali indipendenti dal protocollo e dalla piattaforma) che estendono la funzionalità di un server HTTP, o più in generale di un server Web, quando questo sia stato abilitato a Java. Le servlet sono simili alle applet nel senso che sono oggetti compilati sotto forma di bytecode che possono essere caricati sulla rete dinamicamente, ma, contrariamente alle applet, agiscono sul lato server, sul quale possono effettuare ogni tipo di operazione (esecuzione di altri programmi, input/output su file, accesso a dati residenti sul server, a database, etc.), e generare, con i risultati così ottenuti ed elaborati, un contenuto dinamico per la pagina HTML che viene spedita al cliente in risposta ad una richiesta. Diversamente dalle applet, esse non sono dotate di un'interfaccia grafica (cioè sono, come si dice, oggetti *faceless*) in quanto non devono essere visualizzate direttamente in nessun posto, ma devono solo creare dinamicamente un contenuto sotto forma di uno stream di formato HTML che il server rispedisce al cliente, il cui browser lo visualizza, esattamente come fa per le pagine HTML statiche.

Le servlet funzionano su un modello di ricezione di richieste e generazione di risposte. I dati risultanti da un'elaborazione, richiesti dal cliente su una pagina HTML per mezzo di una serie di parametri raccolti e trasmessi da un normale form HTML mediante un'operazione di tipo POST o GET attraverso lo stream della richiesta, sono restituiti dal server, dopo il processo di produzione, mediante una generazione dinamica dello stream della risposta.

Le servlet possono ricevere i dati di una richiesta, oltre che da un form, anche da uno stream di input prodotto da un'applet (scambio di dati applet/servlet mediante la serializzazione di oggetti, su cui ci soffermeremo in seguito).

1.4.3 Richiamo alla tecnica dei CGI

Ricordiamo che la tecnica dei CGI (*Common Gateway Interface*) consente l'esecuzione di programmi (detti appunto CGI) su un server Web in seguito ad una spacifica richiesta di un cliente sulla rete, generalmente tramite un form. Fare qui un richiamo a questa tecnica non è affatto inutile per varie ragioni: innanzi tutto perché i CGI sono i precursori delle servlet ed il loro concetto è stato esteso a queste poi perché la tecnica è analoga ed anche perché si può sempre presentare l'occasione anche per chi sviluppa applicazioni server Java di dover fare ricorso per qualche ragione (ad esempio quando si dispone di programmi già compilati) ai CGI che sono sempre supportati da ogni server Web Java.

I server Web, che prevedono di avere i file HTML in una sottodirectory della radice del server chiamata generalmente `htdocs`, riservano ai CGI un'altra sottodirectory della radice chiamata di solito `cgi-gin` (o in sue sottodirectory):

```
                httpd
        htdocs          cgi-bin
```

Il CGI deve far precedere il suo contenuto di output da una riga:

```
content type=text/tipo contenuto
```

seguita obbligatoriamente da una riga vuota (in C dai caratteri `\n\n` inclusa l'andata a capo della riga precedente)

Esempio con link, form, #exec, programma.

in servlet vedi idem

e linee di comando

e aanche programma Java standalone testo che chiama servlet

e anche con argomenti

1.4.4 Un esempio di un programma CGI

Un esempio di programma CGI, scritto in linguaggio C, della massima semplicità può essere il seguente:

```
#include <stdio.h>

  void main(int c, char *v[]) {
     printf("Content-type: text/html\n\n");          /* scritta obbligatoria */
     printf("Salve, gente!");
 }
```

Questo programma, compilato in un eseguibile di nome cg, messo sul server e lanciato da un form come il seguente:

```
<HTML>
<body>
<form action="/cgi-bin/SPAGNA/cg" method=POST>
<input type=submit value="vai"></form>
</body>
</HTML>
```

genera sul browser del cliente una pagina come quella di figura 11.1.
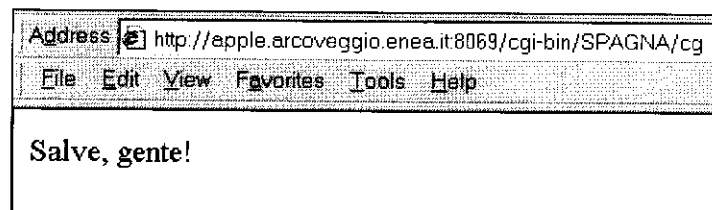
Figura 11.1 Pagina Web prodotta dal CGI dell'esempio.

1.4.5 Programmi CGI e servlet

Le servlet si pongono come un'alternativa valida, se non superiore, ai programmi CGI, che prima dell'avvento di Java erano stati il modello dominante per le applicazioni server in sistemi client/server sul Web.

Le servlet presentano infatti rispetto ai programmi CGI vari vantaggi, tra i quali sono da annoverare:

- le applicazioni server Java tipo servlet, una volta inizializzate, non aprono un nuovo processo ad ogni richiesta proveniente da un client, come avviene invece per i CGI, che devono quindi riaprire ogni volta anche le eventuali connessioni ad esempio a database. Una servlet rispondente a varie richieste, o anche più servlet concorrenti diverse, possono funzionare su vari thread in parallelo all'interno dello stesso processo del server che resta attivo come un demone; poiché ogni invocazione di una servlet apre un thread (processo leggero) anziché un processo (pesante) le prestazioni sono migliori e cioè nettamente più veloci e richiedenti meno memoria;

- mentre i programmi CGI sono, come si dice, *stateless*, cioé non conservano il loro stato tra un'esecuzione e l'altra, le servlet, una volta invocate, restano in memoria indefinitamente, pronti a rispondere ad altre chiamate (vengono chiusi solo con un'azione deliberata del server Web) e conservano quindi il loro "stato" tra un'invocazione e l'altra e perciò non devono essere inizializzate ogni volta e possono mantenere connessioni sempre aperte con sorgenti di dati, ad esempio database;

- l'efficienza delle servlet è migliore di quella dei CGI analogamente a quella che si ottiene per altra via con i sistemi tipo ISAPI/NSAPI: nei confronti di questi ultimi le servlet, anche se non proprio altrettanto efficienti, hanno il vantaggio della portabilità delle applicazioni;

- la sicurezza nei programmi CGI richiede che su di essi vengano fatti dei controlli e spesso essi sono obbligati a funzionare all'interno di un ambiente protetto (*sandbox* o *r-box*): i problemi di sicurezza con le servlet possono essere gestiti in modo molto più semplice in quanto le servlet hanno già una sicurezza intrinseca dovuta alla loro natura Java (soggette quindi ad un Java *security manager*);

- poiché per i CGI non c'è un linguaggio standard e i programmi CGI sono generalmente scritti nei più vari linguaggi di programmazione (principalmente C, C++, Perl, Rexx, tcl, csh), per gli amministratori di siti Web si presenta un problema di manutenzione: l'adozione di un unico linguaggio come il Java razionalizzerebbe la gestione di un sito;

- i programmi CGI presentano maggiori problemi per la messa a punto (debug);

- in conclusione le servlet sono indipendenti dalla piattaforma, persistenti, e possono profittare di tutti i tipi di caratteristiche avanzate di Java, compresa la sicurezza, un accesso agevole ai database, e una molto più facile integrazione con le applet Java.

[11.] da William Crawford, Developing Java Servlets,
http://webreview.com/97/10/10/feature/main.html

1.4.6 Servlet usate come Server-Side Include

1.4.6.1 Richiamo ai Server-Side Include (SSI)

Ci sarà utile qui ricordare che il **Server-Side Include (SSI)** è una tecnica dei server Web classici mediante la quale certi particolari comandi (detti ***direttive SSI***) inseriti in una normale pagina HTML sono riconosciuti dal server Web al momento in cui esso predispone la pagina da inviare al cliente e sono dinamicamente rimpiazzate con il contenuto da essi generato sul server (una parte della pagina viene creata dinamicamente dal server che inserisce nel punto della pagina dove c'è il richiamo SSI l'output di un programma eseguito). Si tratta di una tecnica sempre meno usata nella sua forma classica in quanto le viene preferita quella del CGI poichè il numero di cose che si possono fare con essa è limitato e in più le prestazioni sono scarse. Per evitare che il server debba essere costretto ad analizzare tutte le pagine nel tentativo di

controllare la presenza su di esse di istruzioni SSI si può imporre ad esso di limitare questa ricerca ai soli file portanti l'estensione particolare **.shtml** o **.shtm**.

I comandi SSI sono convenzionalmente posti tra i due segni che racchiudono i commenti in HTML, e cioè **<!--** e **-->**. Le più comuni direttive SSI sono **#include** e **#exec** scritte nella forma:

```
#include="nomefile"
```

che include il testo di un determinato file nel posto del comando sulla pagina HTML, e

```
#exec cgi="nomeprogramma"
```

che esegue in modalità CGI un programma (compilato o script) e inserisce sulla pagina HTML il risultato da esso prodotto (quello scritto dal programma sull'output standard).

Per esempio una pagina HTML che richiami un programma CGI, ad esempio un contatore, può essere scritta così:

```
<HTML>
<BODY>
<H3>Pagina chiamata</H3>
Sei l'utente n.
<!--#exec cgi="/cgi-bin/contatore.exe"-->
</BODY>
</HTML>
```

1.4.6.2 L'SSI con le servlet

Un tipo particolare di Server-Side Include evoluto molto promettente è rappresentato dalle servlet richiamate mediante il tag **<servlet>** in una pagina HTML nella forma seguente:

```
<servlet name=miaServlet>
</servlet>
```

Alla servlet possono essere passati degli argomenti nel seguente modo:???#

```
<servlet name=miaServlet>
   <param name=nome1 value=valore1>
   <param name=nome2 value=valore2>
</servlet>
```

```
<servlet code="miaServlet"
    param1="valore1"
    param2="valore2">
</servlet>
```

In corrispondenza di questo *tag* la servlet è caricata, inizializzata ed eseguita con il valore dei parametri specificati e l'output viene inserito in questo punto della pagina al momento in cui essa viene inviata al client.

Esempio#

1.4.7 Java Servlet Development Kit (JSDK)

1.4.7.1 Librerie e ambienti di sviluppo

L'**API Java Servlet** per la creazione di servlet è una libreria standard di estensione Java (Standard Java Extension API), cioè una libreria di classi che non fa parte di quel nocciolo (*core*) del *framework* Java che deve essere sempre necessariamente presente in ogni ambiente Java, ma ne costituisce un'estensione per un uso specifico. Questa API è supportata dalla maggior parte dei server Java attuali.

Il **Java Servlet Development Kit** (**JSDK**) di Javasoft é un ambiente di sviluppo per le servlet, che comprende anche un *Servlet Runner* con il quale si possono provare le servlet nella fase di sviluppo anche senza un vero e proprio Java Web Server funzionante.

1.4.7.2 Gli oggetti servlet e l'interfaccia `Servlet`

Dal punto di vista del linguaggio una servlet é una classe Java che implementa l'interfaccia **`Servlet`** del package `javax.servlet`, e in quanto tale è capace di ricevere uno stream di input da un client che la invoca e di rispondergli con uno stream di output, dopo aver eseguito le operazioni in essa previste.

In particolare nel package `javax.servlet.http` dell'API Java Servlet ci sono le classi **`GenericServlet`** che implementa l'interfaccia `Servlet` e **`HttpServlet`**, da essa derivata, che incapsula le funzionalità server del protocollo HTTP per processare richieste di tipo GET e POST, che vedremo al paragrafo 11.4.8.3.

Quando si definisce una nuova servlet come un'estensione della classe `GenericServlet` bisogna ridefinire i metodi propri dell'interfaccia `Servlet`, e tra questi innanzi tutto il metodo:

```
service(ServletRequest req, ServletResponse res)
```

che è il metodo principale che viene invocato automaticamente dal server Web ogni volta che la servlet é chiamata, nel quale viene racchiusa tutta l'attività di processamento di una richiesta di un cliente.

Questo metodo ha un primo parametro di tipo `ServletRequest`, che incapsula i dati provenienti dal client come richiesta (é il solo punto di entrata dati nella servlet), ed un secondo parametro di tipo `ServletResponse` per la risposta, La lettura della richiesta e la scrittura della risposta viene fatta da e sugli stream relativi a questi due oggetti `InputStream` e `OutputStream` rispettivamente (chiamiamoli ad esempio `req` e `res`), che sono ricavati rispettivamente con i metodi `req.getInputStream()` e `res.getOutputStream()`.

Una servlet può trattare in modo concorrente varie richieste provenienti da diversi clienti perché ciascuna richiesta attiva automaticamente un suo proprio metodo `service()` in un nuovo thread, e questo apre molte possibilità per la creazione di applicazioni collaborative in

cui i client possono condividere gli stessi dati se definiti come variabili d'istanza globali, ma per contro questi tipi di variabili vanno assolutamente evitate quando i clienti non devono condividerle.

Siccome poi tutte le servlet sono eseguite in un'unica JVM del server, le diverse servlet potrebbero anche condividere dei dati.(vedi#)

Il server, prima di mandarle in esecuzione, inizializza automaticamente le servlet invocandone il metodo `init(ServletConfig)`, un altro metodo proprio dell'interfaccia `Servlet`, che è in generale riscritto dal programmatore, il quale vi pone le operazioni iniziali che non devono più essere ripetute ad ogni richiesta. Infatti il metodo `init()` fa le inizializzazioni una volta per tutte alla prima invocazione della servlet e queste restano valide alle successive invocazioni perché, come è stato detto, una volta lanciata, la servlet resta attiva e non deve ripartire ogni volta che viene invocata da un client.

La vita delle servlet si chiude quando viene chiamato dal server il loro metodo `destroy()`, un altro metodo dell'interfaccia `Servlet`, che le predispone alla *garbage collection* per disallocare la memoria da esse non più occupata.

1.4.7.3 Classe `ServletRequest`

La classe **`ServletRequest`** incapsula tutti i dati della richiesta proveniente al server dal cliente che ha invocato la servlet e che il server passa come argomento al metodo `service()` di quello.

Per avere informazioni sull'ambiente in cui è stata avanzata una richiesta ci sono i metodi di accesso dell'oggetto `ServletRequest` passato:

`getServerName()`	restituisce l'*hostname* del server
`getServerPort()`	restituisce il numero di porta del server
`getRemoteAddr()`	restituisce l'indirizzo IP del client
`getRemoteUser()`	restituisce il *remote username*
`getQueryString()`	restituisce la *query string*
`getMethod()`	restituisce il tipo di richiesta (get o post)

Esempio#

1.4.7.4 Classe `ServletResponse`

La classe **`ServletResponse`** incapsula i dati della risposta che il server invia al cliente.

Metodi:#

`ge()`	restituisce
`ge()`	restituisce

1.4.8 Servlet chiamata dalla linea di comando del browser

Per lanciare una servlet da un browser si potrà chiamarla da un file HTML come si è visto sopra o anche direttamente come URL dalla linea di comando del browser con:

```
http://sitoServlet.it:8080/servlet/HelloServlet
```

Una servlet può essere invocata da un cliente mediante un URL che contiene il nome della classe che la rappresenta:

```
http://server.it:8080/servlets/miaservlet
```

ed eventualmente con l'aggiunta di una stringa di query per il passaggio di parametri.

Esempio#

1.4.8.1 Esempio di `GenericServlet` chiamata da un form

Facciamo qui un esempio elementare di servlet generica (`GenericServlet`) che si limita ad inviare al client una stringa di `"Buongiorno"` che comparirà come tale sul browser chiamante. Per compilare il programma con `javac` precisiamo il classpath relativo al package `servlet.jar` con:

```
set classpath=c:\jdk1.2\lib\ext\servlet.jar
```

Segue il codice volutamente semplificato al massimo:

```
// K02serv054.java (F.Spagna) Esempio di GenericServlet chiamata da un form
// 01-16.10.98  (inizio: 14 ottobre 1998)

import java.io.*;
import javax.servlet.*;

public class K02serv054 extends GenericServlet  {

   public void service(ServletRequest req, ServletResponse res)
                        throws ServletException {
      String nome = req.getParameter("nome");
               try {
      PrintStream ps  = new PrintStream(res.getOutputStream());
      ps.println("<b>Salve " + nome);
               } catch(IOException e) {}
   }
```

```
}
```

Questa servlet può rispondere alla richiesta di un form sul file HTML del tipo:

```
<form action=/servlet/serv054>
      <input type=textfield name=nome>
      <input type=submit value="esegui servlet">
</form>
```

il cui aspetto nella pagina HTML presentata al cliente per la richiesta è quello di figura 11.2:
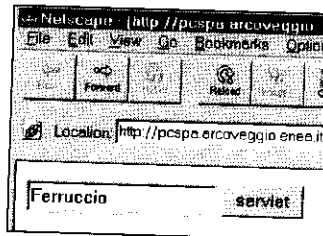
Figura 11.2 Form HTML il cui azionamento chiama una servlet.

ed in tal caso il server, in seguito all'esecuzione della servlet, restituisce al cliente la pagina di risposta che si vede qui di seguito in figura 11.3:
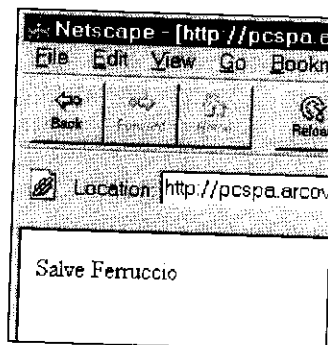
Figura 11.3 Pagina di risposta della servlet `K02serv054` chiamato da un form.

1.4.8.2 Esempio di `GenericServlet` chiamata da un'applet

E' possibile anche far sì che la richiesta ad una servlet venga inviata da un'applet posta su una pagina HTML anziché da un form: in tal caso è sufficiente che l'applet mandi la stessa stringa di query che sarebbe inviata da un form, costituita da una serie di coppie `nome=valore` separate da un segno &, cioè nella forma:

```
/server.it/servlet/servlet1?nome="Mario"&cognome="Musi"
```

Con un'applet chiamata sul file HTML con:

```
<applet code=appK054.class
width=480 height=80></applet>
```

Il codice dell'applet potrebbe essere:

```
// K03appl054.java (F.Spagna) Applet che comunica con la GenericServlet
// K02serv054

import java.io.*;
import java.awt.*;
import java.net.*;

public class K03appl054 extends java.applet.Applet {

   TextField TFdomanda;                                    // campo di input
   Label lab;                                  // etichetta per il risultato
   Button bot;                                        // bottone di comando
   String sitoServ = "http://pcspa.arcoveggio.enea.it:8080";

   public void init() {
      add(TFdomanda = new TextField("", 20));                  // campo input
      add(bot = new Button("servlet"));                     // bottone comando
      add(lab = new lab("              ")); //etichetta che mostra il risultato
   }
                                  // chiama la servlet e ne espone i risultati
   public boolean action(Event ev, Object arg) {
      if (ev.target instanceof Button) {       // se evento di clic sul bottone
            try {
         String nome = TFdomanda.getText(); // legge stringa in campo di input
         URL url = new URL(sitoServ + "/servlet/K02serv054?nome=" + nome);
         InputStream is = url.openStream();      // apertura della connessione
         DataInputStream dis = new DataInputStream(is); // dati su connessione
         lab.setText(dis.readLine());              // legge e mostra la risposta
            } catch (IOException exc) { }     // caso di errore input/output
         return true;
      }
      return false;
   }
}
```

vedi linea di comando risultante#

Il risultato nel caso dell'applet è presentato in figura 11.4.
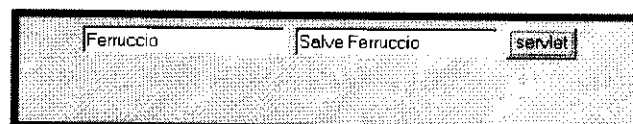
Figura 11.4 Applet che colloquia con una servlet.

1.4.8.3 La classe **HTTPServlet**

La classe **HTTPServlet** rappresenta una servlet HTTP. Nelle servlet HTTP i dati della richiesta e della risposta sono sempre forniti come dati di formato MIME (*MIME#*). La servlet legge tramite lo stream di input e scrive sullo stream di output i dati codificati nel formato previsto: questi due stream possono trasportare dati di qualunque formato, come HTML o diversi formati di immagine, e questo permette la lettura di dati (anche grafici) di qualsiasi forma e la scrittura della risposta anche in forma qualsiasi.

Il metodo **doGet()**: quando un form HTTP accede ad una servlet HTTP usando il metodo GET con il quale tutta l'informazione è codificata (*encoded*) nella variabile d'ambiente QUERY_STRING.(pag.264)#

Il metodo **doPost()**: qu

Ecco un esempio di una servlet HTTP:

```
// K04miaServlet.java (F.Spagna) Esempio di servlet HTTP
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class K04miaServlet extends HttpServlet {

   public void doGet(HttpServletRequest ric, HttpServletResponse ris)
                  throws ServletException, IOException {
      ServletOutputStream sos = ris.getOutputStream();
      ris.setContentType("text/html");
      sos.println("<HTML><HEAD><TITLE>Mio Servlet</TITLE></HEAD><BODY>"
               + "<H3>Titolo</H3>"
               + "<P>Testo dell'HTML"
               + "</BODY></HTML>");
      sos.close();
   }
   public String getServletInfo() {
      return "Esempio di servlet";
   }
}
```

Una servlet così definita invia al cliente la pagina HTML di figura 11.4.

Figura#

Figura 11.4 Pagina sul browser del cliente ricevuta da una semplice servlet HTTP.

1.4.9 Passaggio di oggetti anziché stringhe tra applet e servlet

Tra le applet e le servlet, oltre che le stringhe viste ai paragrafi precedenti, possono essere passati nei due versi anche degli oggetti serializzati.

Riprendendo l'esempio della servlet K02serv054 e dell'applet K03appl054 dei paragrafi 11.4.6.1 e 11.4.6 il codice può essere modificato come segue, prima per la servlet:

```
// K05serv054.java (F.Spagna) Esempio di GenericServlet chiamata da un form
// 01-16.10.98  (inizio: 14 ottobre 1998)

import java.io.*;
import javax.servlet.*;

public class K05serv054 extends GenericServlet  {

   public void service(ServletRequest req, ServletResponse res)
                       throws ServletException {
      String nome = req.getParameter("nome");
             try {
      PrintStream ps  = new PrintStream(res.getOutputStream());
      ps.println("Salve " + nome);
// ObjectOutputStream oos = new ObjectOutputStream(res.getOutputStream());
   // oos.writeObject("<b>Salve " + nome);
   // oos.flush();
             } catch(IOException e) {}
   }
}
```

Il codice dell'applet sarebbe:

```
// K05appl054.java (F.Spagna) Applet che comunica con la GenericServlet
// K05serv054

import java.io.*;
import java.awt.*;
import java.net.*;

public class K05appl054 extends java.applet.Applet {

   TextField TFdomanda;                                     // campo di input
   Label lab;                                  // etichetta per il risultato
   Button bot;                                         // bottone di comando
   String sitoServ = "http://pcspa.arcoveggio.enea.it:8080";

   public void init() {
      add(TFdomanda = new TextField("", 20));                 // campo input
      add(bot = new Button("servlet"));                    // bottone comando
      add(lab = new lab("                 ")); //etichetta che mostra il risultato
   }
                                 // chiama la servlet e ne espone i risultati
   public boolean action(Event ev, Object arg) {
      if (ev.target instanceof Button) {       // se evento di clic sul bottone
            try {
         String nome = TFdomanda.getText(); // legge stringa in campo di input
         URL url = new URL(sitoServ + "/servlet/K02serv054?nome=" + nome);
         InputStream is = url.openStream();      // apertura della connessione
         DataInputStream dis = new DataInputStream(is); // dati su connessione
         lab.setText(dis.readLine());            // legge e mostra la risposta
  // ObjectInputStream ois = new ObjectInputStream(is);
      //            try {
      // String oggStr = (String)(ois.readObject());
      // lab.setText(oggStr);                                // scrive risposta
      //            } catch (ClassNotFoundException cnfe) { }
            } catch (IOException exc) { }      // caso di errore input/output
         return true;
      }
```

```
        return false;
    }
}
```

1.4.10 Comunicazioni applet-servlet

Un'applet può passare dei parametri ad una servlet in diversi modi, per esempio aprendo una connessione URL (`URLConnection`) alla servlet e passando i dati con un'operazione POST, oppure aprendo una connessione socket alla servlet o anche aprendo una connessione RMI.

Un esempio di un'applet che interroga una servlet e ne scrive i risultati che riceve è fornito dalla Sun stessa in ...#. In esso si definisce l'applet base presente sulla pagina dell'utente ed un'altra classe particolare ausiliaria che incapsula la chiamata della servlet e la ricezione dei risultati. Questa classe, chiamata nell'esempio `ServletMessage` ha come parametro del construttore l'URL della servlet e possiede due metodi dei quali il primo è:

```
InputStream sendMessage(Properties args)
```

che prende come argomento un oggetto di classe `Properties` (praticamente un'`Hashtable`) e che restituisce un `InputStream`, ed il secondo metodo:

```
String toEncodeString(Properties args)
```

che codifica gli argomenti come una stringa formata da coppie di nomi-valori (query), nel formato voluto dalla servlet.

L'applet istanzia un oggetto di questo tipo inizializzato con l'URL della servlet da chiamare e ne invoca il metodo `sendMessage()` passandole l'`Hashtable` (più particolarmente le `Properties`) di coppie nomi-valori definita ed è il valore restituito dal metodo che costituisce l'`InputStream` che l'applet legge come `DataInputStream`.

Facciamo un esempio che riguarda un'applet che chiede al cliente un nome in un campo di input e lo passa ad una servlet che risponde restituendo il numero che corrisponde a quel nome in una sua tabella interna. Ecco il codice lato applet (si noti in particolare il metodo per codificare gli argomenti come stringa di coppie `name=value` separati dal segno `&`):

```
// K05client053.java (F.Spagna) Parte client di una comunicaz. applet/servlet

import java.io.*;
import java.awt.*;
import java.util.*;
import java.net.*;

public class K05client053 extends java.applet.Applet {

   Label tit = new Label("Ricerca di numero");                  // etichetta titolo
   Label lab = new Label("                  ");          // per mostrare i risultati
   TextField TFdomanda;                                          // campo di input
   Button bot;                                     // bottone di comando ricerca
   String sitoServA = "http://pcspa.arcoveggio.enea.it:8080";

public void init() {
```

```
      tit.setFont(new Font("Helvetica", Font.BOLD, 18));      // font titolo
      add(tit);
      add(TFdomanda = new TextField("", 20));
      add(lab);
      add(bot = new Button("Cerca"));
   }
   public boolean action(Event ev, Object arg) {
      if (ev.target instanceof Button) {      // se evento di clic sul bottone
             try {
         String nome = TFdomanda.getText(); // legge stringa su campo di input
         Properties par = new Properties();    // Hashtable coppie nomi-valori
         par.put("name", nome);               // mette nome chiesto in Hashtable
         String strcod = "?" + encodedString(par);      // argomenti codificati
         URL urlQuery = new URL(sitoServA + "/servlet/servB053" + strcod);
         InputStream is = urlQuery.openStream(); // apertura della connessione
         DataInputStream dis = new DataInputStream(is); // dati su connessione
         lab.setText(dis.readLine());                  // scrive risposta ricevuta
             } catch (IOException e) { }      // caso di errore input/output
      }
   }
   // codifica gli argomenti come stringa di coppie name=value separati da &
   String encodedString(Properties par) {
      StringBuffer sBuf = new StringBuffer();           // stringa modificabile
      Enumeration nomi = par.propertyNames();         // numera tutte le chiavi
      String separ = "";              // valore iniziale del carattere separatore
      while (nomi.hasMoreElements()) {                    // finche' ci sono dati
         String nome = (String)nomi.nextElement();        // va al dato seguente
         sBuf.append(separ + URLEncoder.encode(nome) + "=" +
                             URLEncoder.encode(par.getProperty(nome)));
         separ = "&";              // successivi valori del carattere separatore
      }
      return sBuf.toString();               // restituisce la stringa costruita
   }
}
```

ed ecco il codice lato servlet:

```
// K06servB053.java (F.Spagna) Parte server di una comunicaz. applet/servlet

import java.io.*;
import java.util.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class K06servB053 extends HttpServlet {

   Hashtable telef = new Hashtable();

   public void init() throws ServletException {
      telef.put("Carboni", "3743");
      telef.put("Poli",    "3360");
      telef.put("Spagna",  "3488");              // alcuni valori dell'Hashtable
   }
   public void doGet (HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException {
      init();                          // perche' non parte da solo all'inizio?
      res.setContentType("text/html");       // tipo di dato inviato all'applet
      String nome = req.getParameter("name");        // legge dato di richiesta
      ServletOutputStream out = res.getOutputStream(); //strean della risposta
      if (telef == null)                          // se l'Hashtable non esiste
         out.println("Elenco non disponibile");                    // lo scrive
      else                              // altrimenti (se l'Hashtable esiste)
         if (nome != null) {                // se la richiesta e' arrivata bene
```

```
            String risult = (String)telef.get(nome);    // cerca corrispondenza
            if (risult == null) risult = "non in lista"; //risult.se non trova
            out.println(nome + ": " + risult); //in ogni caso scrive risultato
        }
    }
}
```

Con un'applet che contatti ed interroghi periodicamente una servlet si può avere una pagina HTML che si aggiorna continuamente in tempo reale con dati provenienti da un server che evolvono.

Figura#

1.4.11 Applet che comunica con più servlet in cascata

Una servlet che è chiamata da un'applet può a sua volta chiamare un'altra servlet comunicando con esso tramite i suoi stream di entrata e di uscita. E' possibile così concepire anche una rete complessa di servlet che comunicano con una o più applet client.

Se noi facciamo un esempio di un'applet che, come nel caso dell'esempio del paragrafo precedente, chiede al cliente un nome ma, anziché passarlo direttamente alla servlet finale che lo deve processare, lo passa ad una servlet intermedia (servlet A) la quale si incarica di passarlo alla servlet finale (servlet B) perché questa faccia infine l'operazione richiesta dal cliente, che è quella di trovare il numero che corrisponde a quel nome in una sua tabella, rispetto all'esempio precedente dobbiamo solo aggiungere una servlet intermedia di passaggio dei dati (richiesta in un senso e risposta nell'altro), il cui codice (si noti che in questo caso il metodo per codificare gli argomenti come stringa di coppie `name=value` separati dal segno &, che era stato messo nell'applet, è qui inserito nella servlet che deve passare dei dati ad un'altra servlet) è il seguente:

```
// K07servA053.java (F.Spagna) Servlet intermedia in un sistema a tre
// componenti applet/servlet A/servlet B che si passano i dati

import java.io.*;
import java.util.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;

public class K07servA053 extends HttpServlet {

   String sitoB = "http://pcspa.arcoveggio.enea.it:8080";

   public void doGet (HttpServletRequest req, HttpServletResponse res)
                    throws ServletException {
      String nome = req.getParameter("name");         // legge dato di richiesta
            try {
      Properties par = new Properties(); // Hashtable delle coppie nomi-valori
      par.put("name", nome);                  // mette nome chiesto in Hashtable
      String strcod = "?" + encodedString(par);         // argomenti codificati
      URL urlQuery = new URL(sitoB + "/servlet/K06servB053" + strcod);
      InputStream is = urlQuery.openStream();     // apertura della connessione
      DataInputStream dis = new DataInputStream(is); // dati sulla connessione
      String rigaRisposta = dis.readLine();          // riga da leggere sui dati
```

```
      ServletOutputStream out = res.getOutputStream(); //strean della risposta
      out.println(rigaRisposta);                                     // lo scrive
            } catch (IOException exc) { }    // caso di errore Input/output
   }
    // codifica gli argomenti come stringa di coppie name=value separati da &
   String encodedString(Properties par) {
      StringBuffer sBuf = new StringBuffer();           // stringa modificabile
      Enumeration nomi = par.propertyNames();         // numera tutte le chiavi
      String separ = "";              // valore iniziale del carattere separatore
      while (nomi.hasMoreElements()) {                   // finche' ci sono dati
         String nome = (String)nomi.nextElement();        // va al dato seguente
         sBuf.append(separ + URLEncoder.encode(nome) + "=" +
                             URLEncoder.encode(par.getProperty(nome)));
         separ = "&";             // successivi valori del carattere separatore
      }
      return sBuf.toString();                 // restituisce la stringa costruita
   }
}
```

1.4.12 Upload di un file su un server per mezzo di una servlet

Quando un cliente da un browser nella richiesta ad un server fatta su un form invia un file, per estrarre il file il file dalla richiesta una servlet può usare un codice come quello dell'esempio che segue:

```
// PostMultiServlet.java (F.Spagna) Esempio di POST con multi
// 01-08.09.99 (inizio: 8 settembre 1999)

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PostMultiServlet extends HttpServlet {

   public void init(ServletConfig config) throws ServletException  {
      super.init(config);
   }
   public void service(HttpServletRequest ric, HttpServletResponse ris)
                                   throws ServletException, IOException {
      ris.setContentType("text/html");
      PrintWriter out = ris.getWriter();
      String nomeFile = null;
      if (ric.getMethod().equals("POST")
          && ric.getContentType().startsWith("multipart/form-data")) {
         int indice = ric.getContentType().indexOf("boundary=");
         if (indice < 0) {
            System.out.println("non trova il tipo di boundary");
            return;
         }
         String boundary = ric.getContentType().substring(indice + 9);
         ServletInputStream instream = ric.getInputStream();
         byte[] buf = new byte[8192];
         int lung = 0;
         String riga = null;
         boolean ancoraDati = true;
         lung = instream.readLine(buf, 0, buf.length);
         riga = new String(buf, 0, 0, lung);
         while(riga.indexOf(boundary) < 0 && ancoraDati) {
            lung = instream.readLine(buf, 0, buf.length);
            riga = new String(buf, 0, 0, lung);
            if (riga != null)
```

```
                out.println("<br>input=" + riga);
            if (lung < 0)
                ancoraDati=false;
         }
         if (ancoraDati) {
            lung = instream.readLine(buf, 0, buf.length);
            riga = new String(buf, 0, 0, lung);
            out.println("<br>"+riga);
            if (riga.indexOf("desc") >= 0) {
               lung = instream.readLine(buf, 0, buf.length);
               riga = new String(buf, 0, 0, lung);
               lung = instream.readLine(buf, 0, buf.length);
               riga = new String(buf, 0, 0, lung);
            }
         }
         while (riga.indexOf(boundary) < 0 && ancoraDati) {
            lung = instream.readLine(buf, 0, buf.length);
            riga = new String(buf, 0, 0, lung);
         }
         if (ancoraDati) {
            lung = instream.readLine(buf, 0, buf.length);
            riga = new String(buf, 0, 0, lung);
            out.println("<br>"+riga);
            if (riga.indexOf("filename") >= 0) {
               int indiceIniz = riga.indexOf("filename");
               String path = riga.substring(
                    indiceIniz + 10, riga.indexOf("\"", indiceIniz + 10));
               System.out.println("<br>" + path);
               nomeFile = path.substring(path.lastIndexOf('\\') + 1);
               lung = instream.readLine(buf, 0, buf.length);
               riga = new String(buf, 0, 0, lung);
               out.println("<br>"+riga);
            }
         }
         byte fileBytes[] = new byte[50000];
         int offset = 0;
         if (ancoraDati)
            while(riga.indexOf(boundary) < 0 && ancoraDati) {
               lung = instream.readLine(buf, 0, buf.length);
               riga = new String(buf, 0, 0, lung);
               if (lung > 0 && (riga.indexOf(boundary) < 0)) {
                  System.arraycopy(buf, 0, fileBytes, offset, lung);
                  offset += lung;
               } else
                  ancoraDati = false;
            }
         out.println("<p>bytes uploaded = " + (offset - 4) + " " + nomeFile);
                // trim last two newline/return characters before using data
         ////  for (int i=0;i");
         out.close();
         FileOutputStream fos = new FileOutputStream("files\\" + nomeFile);
         fos.write(fileBytes, 0, offset - 4);
         fos.close();
      }
   }
}
```

Se il form di richiesta sul browser è creato con:

```
<FORM ACTION="/servlet/PostMultiServlet"
    METHOD="POST" ENCTYPE="multipart/form-data">
    <INPUT TYPE="TEXT" NAME="desc" value="">
```

```
    <BR><INPUT TYPE="FILE" NAME="filecontents" value="">
    <BR><INPUT TYPE="SUBMIT" VALUE="salva">
</FORM>
```

e si presenta come in figura 11.xx:
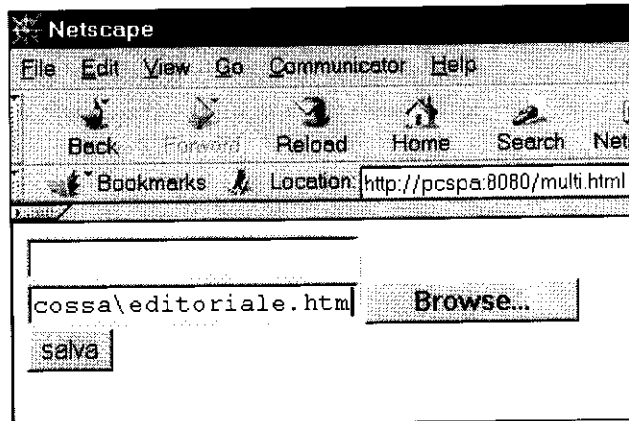
Figura 11.xx Form per fare l'upload di un file su un server.

la servlet legge lo stream di input della richiesta e ne estrae il file e altre informazioni e rispedisce al cliente una pagina come quella di figura 11.xx.
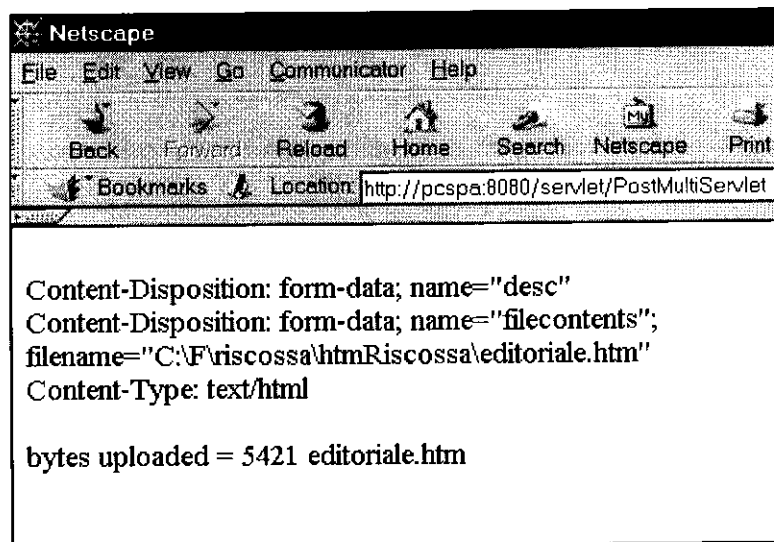
Figura 11.xx Risposta della servlet all'upload dell'esempio.

1.4.12.1 Argomenti di una servlet

Abbiamo visto come alle servlet possono essere passati degli argomenti tramite coppie di stringhe `nome=valore` nella stessa invocazione della servlet nella posizione prevista per la stringa di query dell'URL.

Ma nel Java Web Server alle servlet possono essere passati degli argomenti anche mediante un file `miaServlet.initArgs` posto nella stessa directory della servlet.

L'amministratore del server ha la facoltà di passare suoi argomenti alle servlet al momento del loro caricamento e inizializzazione, specificandoli nella configurazione del server.

1.4.12.2 Sessione di un cliente sul server

Quando uno stesso cliente può chiamare diverse servlet sullo stesso server e si vuole tenere traccia delle sue caratteristiche o dei risultati di certe operazioni da lui fatte passando da una servlet all'altra, si può stabilire e attribuire una **sessione** a quel cliente che mantiene lo "stato" del cliente

Viene qui fatto un esempio molto semplice con due servlet A e B: la servlet A viene chiamata prima, stabilisce una sessione per il cliente chiamante, legge l'ora (è un esempio come un altro) e la memorizza a livello di sessione come un dato riservato al cliente: quando viene invocata dallo stesso cliente la servlet B può leggere questo dato riferendosi all'identificatore di sessione e lo pubblica sulla pagina di risposta.

Ecco il codice per la servlet A:

```
// K08sessionServA.java (F.Spagna) Sessione di un cliente su un server

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class K08sessionServA extends HttpServlet {

   public void doGet (HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
      HttpSession sessione = req.getSession(true);
      sessione.putValue("ora", "dati: " + (new java.util.Date()));
      ServletOutputStream out = res.getOutputStream();
      out.println("<HTML>" + " sessione : " + sessione
                       + "<br> sessionId: " + sessione.getId());
   }
}
```

e quello per la servlet B:

```
// K09sessionServB.java (F.Spagna) Sessione di un cliente su un server

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class K09sessionServB extends HttpServlet {

   public void doGet (HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
      HttpSession sessione = req.getSession(true);
```

```
        String s = (String)sessione.getValue("ora");
        ServletOutputStream out = res.getOutputStream();
        out.println("<HTML>" + " sessione : " + sessione
                             + "<br> sessionId: " + sessione.getId()
                             + "<br> " + s);
    }
}
```

Nella figura seguente 12.5 sono riprodotte le pagine HTML che queste due servlet inviano allo stesso cliente che le ha chiamate in successione, nelle quali si vede che il numero identificatore della sessione del cliente è lo stesso e che lo stato del parametro "ora" persiste:
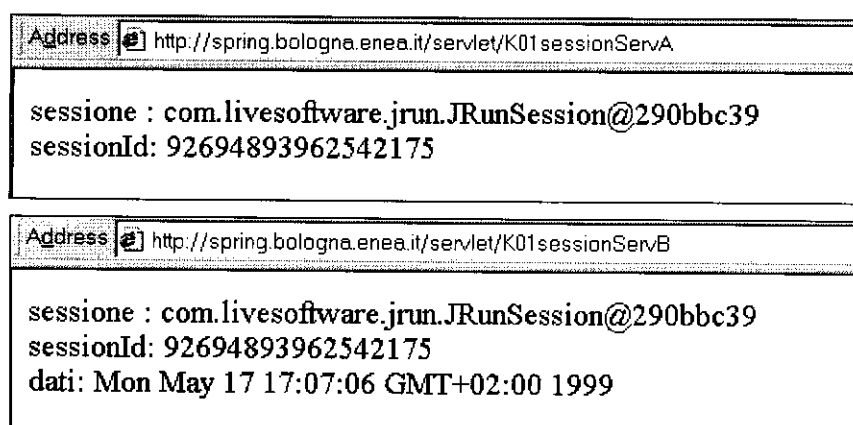
Figura 11.5 Pagine Web prodotte da due servlet chiamate da uno stesso cliente.

1.4.13 Prodotti Sun per le servlet

La Sun Microsystems ha sviluppato una famiglia (*suite*) di prodotti volti all'utilizzo di Java sui server Web, costituita per il momento da:

- il Java Servlet Development Kit (JSDK)
- il Java Web Server
- il Java Server Toolkit
- il Java NC Server

Alla base di tutti questi prodotti c'è la **Java Server API**, che è una libreria di classi per la creazione di soluzioni Java server *cross-platform*.

1.4.14 Il Java Web Server

1.4.14.1 Caratteristiche

Il **Java Web Server** (inizialmente chiamato *Jeeves*) di Sunsoft, che fa parte della suite di cui al paragrafo 11.4.5. supporta la Java Servlet API costituita da classi che permettono agli sviluppatori di siti Web di incorporare nelle pagine un contenuto dinamico generato da applicazioni eseguite sul server. Essendo il Java Web Server scritto interamente in Java, è

indipendente dalla piattaforma (è cioè, come si dice, *cross-platform*). Esso viene rilasciato attualmente in una versione beta (la versione 1.1), che è già abbastanza completa, e richiede sulla macchina la versione finale del JDK 1.1.

Il server supporta controlli di accesso user/group, Secure Sockets Layer (SSL) e i proxy (disk caching).

I compiti dell'amministratore vengono svolti tramite client (browser) del server stesso (server amministrativo alla porta 9090).

1.4.14.2 CGI nel Java Web Server

Il Java Web Server permette l'esecuzione di applicazioni CGI classiche, ma le prestazioni in tal caso non sono eccellenti.

1.4.14.3 Installazione

La versione 1.1 beta di Java Web Server può essere scaricata dal sito:

http://www.javasoft.com/products/java-server/

sotto forma di file `jws11-win32-gl-ssl.exe` di 7578 kB. L'esecuzione del file installa rapidamente il Java Web Server in una directory "JavaWebServer1.1" con le sottodirectory:

admin, bin, cgi-bin, lib, logs, properties, public_html, realms, servlets, srcdemos, system.

L'avvio del server avviene con il programma eseguibile `httpd.exe` situato nella sottodirectory `bin`.

1.4.15 Le servlet sul Java Web Server di Sun

Le servlet installate su un Java Web Server di Sun, per essere riconosciute dal server stesso devono essere posti nella sottodirectory `servlets/` sotto la directory radice (*root*) del server. In questo modo essi sono considerati affidabili (*trusted*) e possono effettuare ogni operazione sul server (apertura in lettura o scrittura di un file, connessione con un socket esterno, etc.). E' anche possibile richiamare su un server delle servlet remote specificandone l'URL, però in tal caso esse devono essere firmate (*signed*) perché considerate in linea di principio inaffidabili.

Per far riconoscere una servlet al server Java Web Server dall'interfaccia di amministrazione raggiungibile alla porta 9090 (http://sitoServer.it:9090). Facciamo vedere qui di seguito in figura 11.6 le varie schermate che si succedono nell'operazione (la prima di login con entrata dei dati admin/admin, poi quella di scelta del servizio (da scegliere servlets), quindi quella del servizio servlets (scegliere add) ed infine la mascherina per aggiungere la servlet alle altre, con un nome convenzionale e quello del bytecode, non dimenticando di scriverne l'estensione .class).
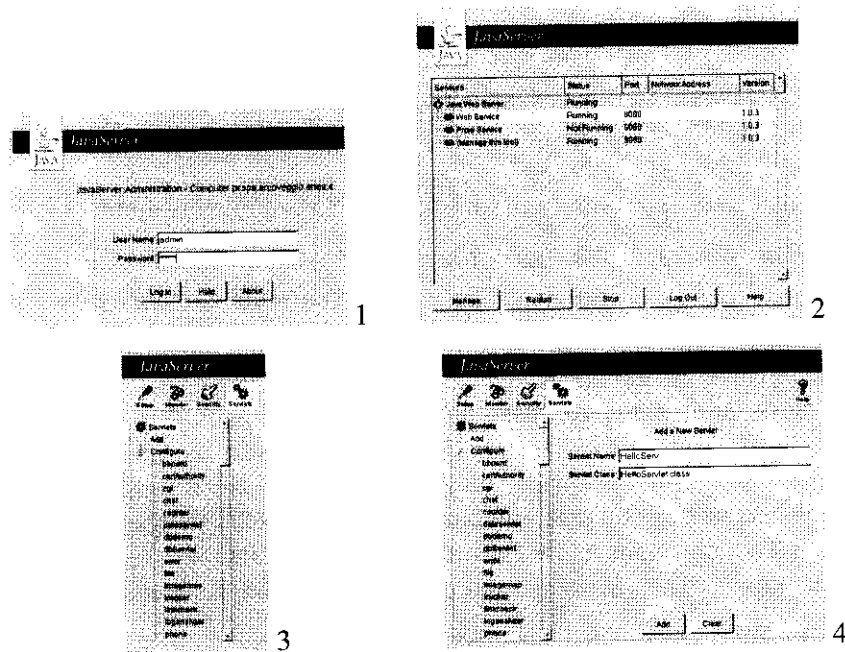
Figura 11.6 Fasi dell'operazione di riconoscimento di una servlet da parte del server JWS.

1.5 Pagine compilate (Java Server Pages)

1.5.1 La tecnica ASP di Microsoft

Prima di descrivere le JSP è utile richiamare rapidamente i concetti della tecnica delle **Active Server Pages (ASP)** di Microsoft per ambiente Windows.

Le pagine ASP sono pagine di formato HTML che contengono una o più porzioni di codice (scritto in un linguaggio di *scripting*) che deve essere eseguito dal server, il quale inserisce al loro posto il risultato del codice (l'*ouput*) che è prodotto come testo in formato HTML standard, così da generare una pagina HTML che viene inviata al cliente, il cui browser la vede nella sua forma finale, indistinguibile dalle altre pagine HTML standard (se non per l'estensione nel nome del file). Il server Web, quando prepara una pagina per un cliente che ne ha fatto richiesta, fa una distinzione tra le pagine HTML statiche normali e le pagine con estensione ASP, che sono quelle che contengono il codice ASP e che sono preparate dinamicamente.

Nel protocollo HTTP non è prevista la conservazione dello stato tra una chiamata e l'altra al server dallo stesso cliente: una volta che il server ha esaudito una certa richiesta perde memoria del richiedente. In ASP questo problema è risolto con il concetto di sessione (variabile `session`): alla prima richiesta di un cliente il server apre per lui e gli riserva una sessione che memorizza le informazioni di stato tra una chiamata e l'altra per tutta la durata della sessione, e cioè fino a che questa non viene esplicitamente chiusa dal server (`abandon`) o dopo che il cliente non ha fatto più richieste per un determinato tempo (20 minuti di default).

L'ASP accetta per gli script diversi linguaggi di *scripting*: VB-Script, JavaScript, REXX, Python, Perl, etc) e all'interno degli script possono essere richiamati, con la tecnica COM, anche componenti software compilati in altri linguaggi.

Le istruzioni ASP VB-Script devono essere contenute tra un simbolo `<%` ed uno `%>`.

La direttiva **`#Include`** consente di inserire intere parti di testo, che possono essere del codice HTML o anche del codice di *scripting*, prelevandole da un file di testo, che può servire diverse pagine, con un risparmio gòobale di testo scritto nelle varie pagine.

I server Web che per il momento supportano questa tecnologia sono principalmente quelli di Microsoft e ciè l'IIS e il PWS.

La tecnologia delle **Java Server Pages (JSP)** è analoga a quella dell'ASP, con la differenza che le pagine dinamiche, contenenti il codice eseguibile sul server, per essere riconosciute dal server stesso devono portare l'estensione JSP, che il codice è scritto in Java (non JavaScript) e che essa è supportata dai server Web Java con un'abilitazione specifica per le JSP.

Per meglio fissare le idee riportiamo qui un semplice script inserito in una pagina che ha il compito di presentare sul browser del cliente una tavola pitagorica, dopo aver ricevuto una variabile (potrebbe essere il nome del cliente, proveniente da un form o digitata direttamente in linea di comando) da una stringa di query concatenata con l'URL della richiesta. Nella pagina è stata inserita una direttiva `#include` di un piccolo file di testo a titolo di esempio. Ecco il testo del file ASP:

Il risultato prodotto dal browser del cliente è visibile in figura 11.7.
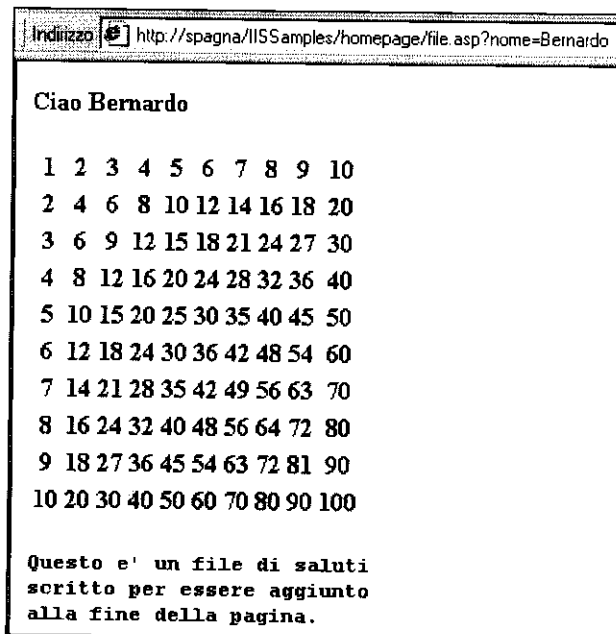
Figura 11.7 Risultato di una pagina ASP dopo l'elaborazione del server.

1.5.2 Java Server Pages (JSP)

Con la tecnologia delle **Java Server Pages (JSP)** il Java Web Server presenta una modalità di funzionamento sul tipo dell'SSI (vedi paragrafo 11.4.4) detta di compilazione di pagina (***Page Compilation***) che consiste nel poter inserire direttamente sulla pagina HTML, in file aventi l'estensione JHTML, una porzione di codice Java, che viene compilato sul momento dal server in una servlet che viene eseguito sul server (da ben distinguere questo codice dal JavaScript presente in una pagina HTML che viene eseguito invece sulla macchina client). Questa tecnica, alternativa e più semplice nell'utilizzazione di quella delle servlet (richiede minori conoscenze nella programmazione di Java), offre una grande flessibilità di uso nella pagina HTML ed è analoga e segue il modello di quella dell'ASP (Active Server Pages) di Microsoft, anche se in questo caso non si tratta di linguaggio di scripting (VBScript o JScript), come è quello dell'ASP, ma di vero e proprio linguaggio Java. Con le JSP si può separare il lavoro di preparazione della pagina HTML da quello della programmazione della logica della servlet, mentre le servlet devono fare le due cose insieme, e si evita inoltre il lavoro manuale di compilazione.

Il codice Java che genera il contenuto dinamico della pagina HTML viene inserito in qualunque posto di essa racchiuso tra i due *tag* speciali **`<java>`** e **`</java>`**.

In una dichiarazione "`java type=class`" possono essere dichiarate classi interne alla servlet o variabili di istanza della servlet. Il codice Java è inserito direttamente nel metodo `service()`.

Ecco un esempio, tratto da quello fornito con il Java Web Server 1.1 stesso e da noi leggermente modificato, che crea e utilizza una classe che funziona da contatore:

```
<! K11jsp.jhtml (F.Spagna) Esempio di Java Server Pages >

<java type="import">
```

```
        javax.servlet.http.*
</java>

<java type="class">
        class contatore {            // classe interna (inner class) alla servlet
           int cont = 0;
           public int conta() { return cont; }
           public synchronized void incrementa () { cont++; }
        }
        contatore c = null;                                   // variabile d'istanza
</java>

<java>
        HttpSession s = request.getSession(true);
</java>

<HTML>
<HEAD><TITLE>Hello</TITLE></HEAD>
<H2>Java Server Pages</H2>
<TABLE><TH BGCOLOR=DDDDFF>Esempio di Page Compilation

<java>
        if (c == null)
           c = new contatore();
        int cont = c.conta();
        c.incrementa();
</java>

<BR>Sei il visitatore numero

<java type=print>
        cont                         // questa espressione è valutata e stampata
</java>

 di questa pagina
</TH></TABLE>
</HTML>
```

La pagina inviata al cliente risultante è mostrata nella figura 11.8.
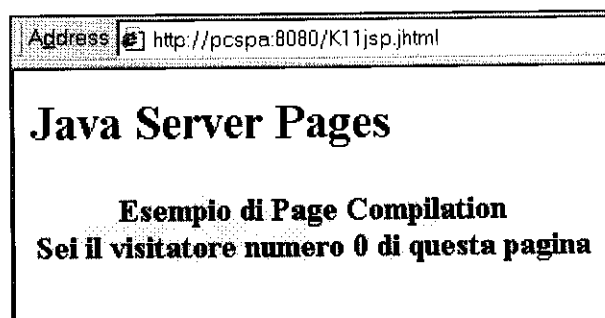
Figura 11.8 Risutato sul browser del cliente della pagina JSP.

Un esempio di pagina JSP con accesso a un database è il seguente, che chiede il cognome dell'utente in una prima pagina HTML normale contenente un form, il quale lancia una pagina JSP che riceve il parametro cognome, cerca il nome corrispondente in un database di persone ed invia al cliente un saluto con il suo nome. L'HTML del form (visibile in figura 11.9) è:

```
<form action="db.jhtml" method=post>
<input type=text name="cognome">
<input type=submit value="saluta">
</form>
```
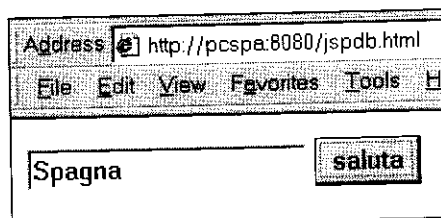
Figura 11.9 Form HTML per chiamare una pagina JSP.

La pagina JHTML è così impostata:

```
<! K01.jhtml (F.Spagna) Esempio di JSP con ricerca su database>
<HTML>

<java type=import>java.sql.*</java>

<h3>Un saluto</h3>

<java>
                try {
   DriverManager.setLogStream(null);
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   Connection con = DriverManager.getConnection("jdbc:odbc:dati");
   String cog = request.getParameter("cognome");
   Statement st = con.createStatement();
   String query = "SELECT * FROM persone WHERE cognome='" + cog + "'";
   ResultSet ris = st.executeQuery(query);
   while (ris.next()) { String nom = ris.getString("nome"); }
   con.close();
   out.println("<p>Ciao, " + nom);
                } catch(ClassNotFoundException e) {} catch(SQLException e) {}
</java>

</HTML>
```

La pagina JSP inviata al cliente è quella della figura 11.10.
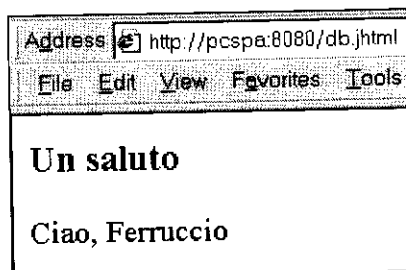
Figura 11.10 Pagina JSP inviata al cliente dopo ricerca del suo nome in un database.

Anche il server Java App Server **Tengah** di WebLogic (*http://www.weblogic.com*), sviluppato interamente in Java, supporta questa funzionalità di Page Compilation. Per maggiori informazioni su di esso vedere:

http://www.internetnews.com/wd-news/1998/06/2501-weblogic.html

1.5.3 Sintassi delle JSP

1.5.3.1 Direttive JSP

Le direttive JSP sono istruzioni, comprese tra i simboli **<%@** e **%>**, che hanno scope su tutto il file JSP usate per stabilire valori globali come definizione del linguaggio di scripting (Java di default), dichiarazione di classi, importazione di classi Java da usare, etc.

Gli attributi stabiliti con una direttiva possono essere:

```
<%@ language="java" %>
<%@ method="doPost" %>
<%@ import="java.io.*" %>
<%@ content_type="" %>
<%@ implements="java.io.Serializable" %>
<%@ extends="" %>
```

La direttiva `include` consente di includere il contenuto di altri file, potendosi così conservare in file separati parti di testo della pagina che si ripetono, come potrebbero essere per esempio le intestazioni.

1.5.3.2 Dichiarazioni

Nelle dichiarazioni, racchiuse dai simboli **<%!** e **%>**, si definiscono variabili e metodi globali accessibili da tutta la pagina.

1.5.3.3 Espressioni

Le espressioni, racchiuse dai simboli **<%=** e **%>**, rappresentano un modo semplice (un'abbreviazione) per scrivere in un punto della pagina il risultato di un'espressione Java senza dover scrivere il codice con il `println()`.

1.5.3.4 Scriptlet

Le scriptlet sono le porzioni di codice Java, comprese tra i due segni **<%** e **%>**, che sono compilate in una servlet dal motore JSP sul server, dove restano come bytecode disponibile per ogni richiesta.

1.5.3.5 Variabili implicite

Il JSP definisce alcune variabili implicite che rappresentano certi oggetti Java particolari delle servlet, che possono essere usati nelle espressioni e nelle scriptlet:

`out` rappresenta un oggetto `javax.servlet.jsp.JspWriter` che dispone del metodo `println()` per l'output nelle scriptlet

`request` rappresenta un oggetto `javax.servlet.http.HttpServletRequest` relativo alla richiesta

`response` rappresenta un oggetto `javax.servlet.http.HttpServletResponse` relativo alla risposta

1.5.4 Le servlet su altri server Web

Vari server Web diversi dal Java Web Server, come in particolare i server Internet Information Server (I.I.S.) di Microsoft, Enterprise e FastTrack di Netscape e l'Apache HTTP Server, supportano le servlet, ma essi vanno opportunamente configurati per questo. SunSoft ha creato un package che può essere usato per introdurre il supporto delle servlet in questi tre server Web: il JSDK stesso ne fornisce i *patch* (o *plug-in*) per l'installazione. Nel caso dell'IIS (2.0 e 3.0) si deve installare una DLL e iscriverla nel *registry*.

Java Servlet Runner (**JRun**) di Live Software è un altro prodotto per il supporto della Java Servlet API da parte dei server Web IIS di Microsoft e Enterprise Server e FastTrack Server di Netscape. A questo riguardo si può andare a vedere l'URL:

http://www.internetnews.com/prod-news/1997/11/2001-live.html

Un software che permette di implementare le servlet Java sul server Web FastTrack Server 3.01 e Enterprise Server 3.0/3.5 (per qualunque piattaforma) è il **WAICoolRunner** di Gefion Software (WAI sta per Web Application Interface) per cui si può consultare l'URL:

http://www.gefionsoftware.com/WAICoolRunner/

Il **ServletExec** di NewAtlanta è un altro prodotto che implementa l'API Java Servlet per l'IIS di Microsoft e i due suddetti server Web di Netscape per varie piattaforme. Per la versione 1.1 è previsto anche il supporto della Page Compilation. Vedere:

http://www.newatlanta.com/

Altri server Web che supportano le servlet sono il Lotus Domino Go Webserver Pro, il Novell Netscape Enterprise Server Pro for NetWare e il WebSite Professional. Ma si prevede che presto tutti i server Web supporteranno le servlet.

Un server Web HTTP scritto interamente in Java che supporta le servlet è il **JigSaw** del Consorzio W3C, che si può trovare al sito:

http://www.w3.org/Jigsaw/

Un altro piccolo server HTTP che supporta le servlet è l'**Acme.Serve** di Jef Poskanzer, disponibile all'URL:

http://www.acme.com/java/software/Acme.Serve.Serve.html

1.6 I Java Bean

I Java Bean sono componenti software scritti in Java. Per componente si intende un'unità software autosufficiente, riutilizzabile, che può essere aggiunto ad un'applicazione (o applet) con un trattamento visuale per mezzo di strumenti di costruzione di applicazioni. I componenti espongono le loro proprietà per mezzo di metodi pubblici ed eventi permettendo la loro manipolazione con mezzi visuali. Perché le caratteristiche siano esposte devono rispettare delle convenzioni di nome. Lo strumento visuale che maneggerà i componenti mostrerà le caratteristiche esposte e permetterà la loro modifica manuale. Un costruttore di applicazioni per componenti tiene i componenti in una palette o toolbox. Quando si vuole utilizzare un componente (bean) disponibile lo si seleziona, lo si trascina in un form, si aggiusta il suo aspetto ed il suo comportamento e si definisce la sua interazione con gli altri componenti (bean), tutto visivamente, senza scrivere una sola riga di codice.

Le caratteristiche di un Bean sono le proprietà, i metodi e gli eventi.

Il processo mediante il quale un Bean builder scopre le caratterisstiche di un Bean è conosciuto con il nome di introspezione.

Le proprietà di un Bean sono quelle caratterisstiche riguardanti l'aspetto ed il comportamento che possono essere stabilite al momento dell'inserimento del bean (design time).

Un bean comunica con gli altri per mezzo di eventi.

La persistenza è quella proprietà che permette ai Bean di avere il loro stato salvato e restaurato se necessario.

1.6.1 Esempio di creazione di un JavaBean

Facciamo ora un esempio di costruzione ed utilizzazione di un JavaBean della massima semplicità.

Si può scrivere in un file di testo chiamato primoBean.java il seguente codice Java:

```
// K01primoBean.java (F.Spagna) Primo semplice esempio di JavaBean

import java.awt.*;
import java.io.Serializable;

public class L01primoBean extends Label implements Serializable {
   public primoBean(){
      setSize(200, 40);
      setBackground(Color.yellow);
      setForeground(Color.blue);
      setAlignment(Label.CENTER);
      setFont(new Font("", Font.BOLD, 16));
      setText("primoBean");
   }
}
```

Con la compilazione fatta con il solito comando:

```
javac primoBean.java
```

si ottiene il file `primoBean.class`.

Si può creare quindi un file manifesto sotto forma di un file di testo chiamato `manif.txt`, contenente le due righe seguenti:

```
Name: primoBean.class
Java-Bean: True
```

Si può quindi creare il file JAR `primoBean.jar` che contiene il file della classe e quello del manifesto, con il comando:

```
jar cfm primoBean.jar manif.txt primoBean.class
```

Dopo aver impostato il CLASSPATH opportuno:

```
set classpath=classes;..\lib\methodtracer.jar;..\infobus.jar
```

si può lanciare con la JVM l'applicativo BdK:

```
java sun.beanbox.BeanBoxFrame
```

Si apre allora l'applicazione Bdk che comincia con una finestra così:
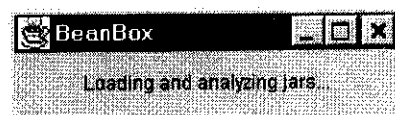
Figura 12.1

e poi quella principale dell'applicazione, che presenta un menù con il quale si può scegliere di fare l'operazione di caricamento di un nuovo Bean:
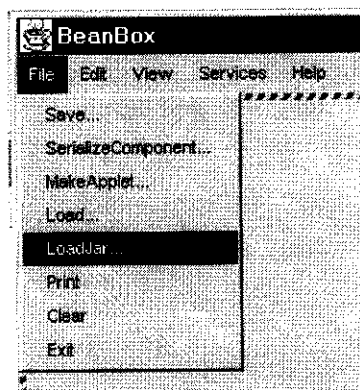
Figura 12.2

Si aprirà allora una finestra di navigazione tra i file nella quale si seleziona il bean primoBean che avevamo creato: a quel punto il nuovo bean compare nella lista dei beans (jar) disponibili in un'altra finestra:
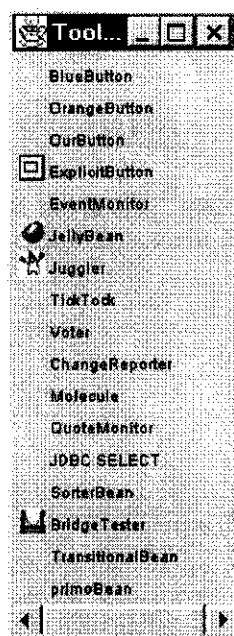
Figura 12.3

Ed a questo punto si può selezionare il primoBean dalla lista e aprire un rettangolo con il mouse trascinato sulla finestra principale e così viene inserito il bean che si può vedere nella figura seguente:
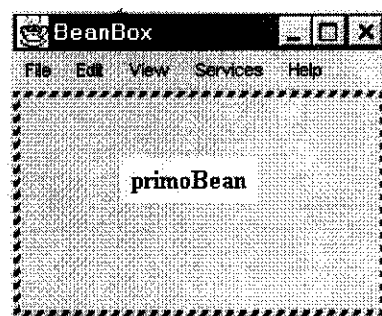
Figura 12.4

che può essere modificato a mano visualmente con una finestra che nel frattempo si è aperta nella quale sono esposte le proprietà del componente modificabili (nel caso nostro sono quelle proprie del componente AWT `Label` da cui la nostra classe deriva):
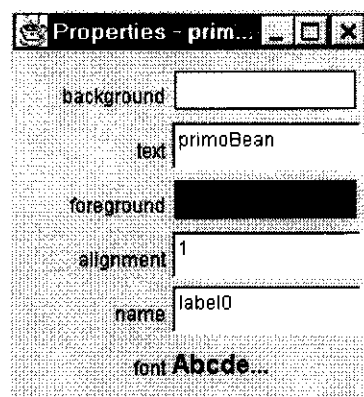
Figura 12.5

1.6.2 Beans Development Kit (BDK)

Il **Beans Development Kit (BDK)** è un'applicazione Java che comprende un ambiente di sviluppo detto **BeanBox** per provare i Java Beans. Esso è disponibile per il momento solo per Windows 95, NT e Solaris.